Deadlock Analysis with Fewer False Positives

Multi-threaded program

```
Thread T1:

sync(G){
  sync(L1){
    sync(L2){}
  }
};
T3 = new T3();
j3.start();
J3.join();
sync(L2){
  sync(L1){}
}
```

```
Thread T2:

sync(G){
  sync(L2){
    sync(L1){}
  }
}
```

```
Thread T3:

sync(L1){
  sync(L2){}
}
```

no yes

Deadlock?

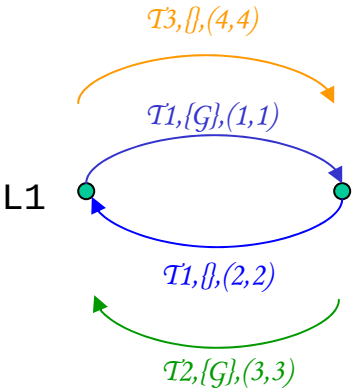
- Program deadlocks between T2 and T3
- Algorithm builds lock graph from trace
- Deadlock potentials show as cycles in graph
- Cycle freedom is by far easier to test

Execute instrumented version of program and extract execution trace

- Old algorithm reports 4 deadlocks, 3 false
- New algorithm only reports 1 (the real one)
- Hence algorithm reduces false positives
- This means less time spent by programmer

```
Lock(T1,G)
Lcck(T1,L1)
Lock(T1,L2)
...
Start(T1,T3)
Lock(T2,G)
Lock(T2,L2)
Lock(T2,L1)
...
Lock(T3,L1)
Lock(T3,L2)
...
Join(T1,T3)
Lock(T1,L2)
Lock(T1,L1)
...
```

Compute graph of lock hierarchies



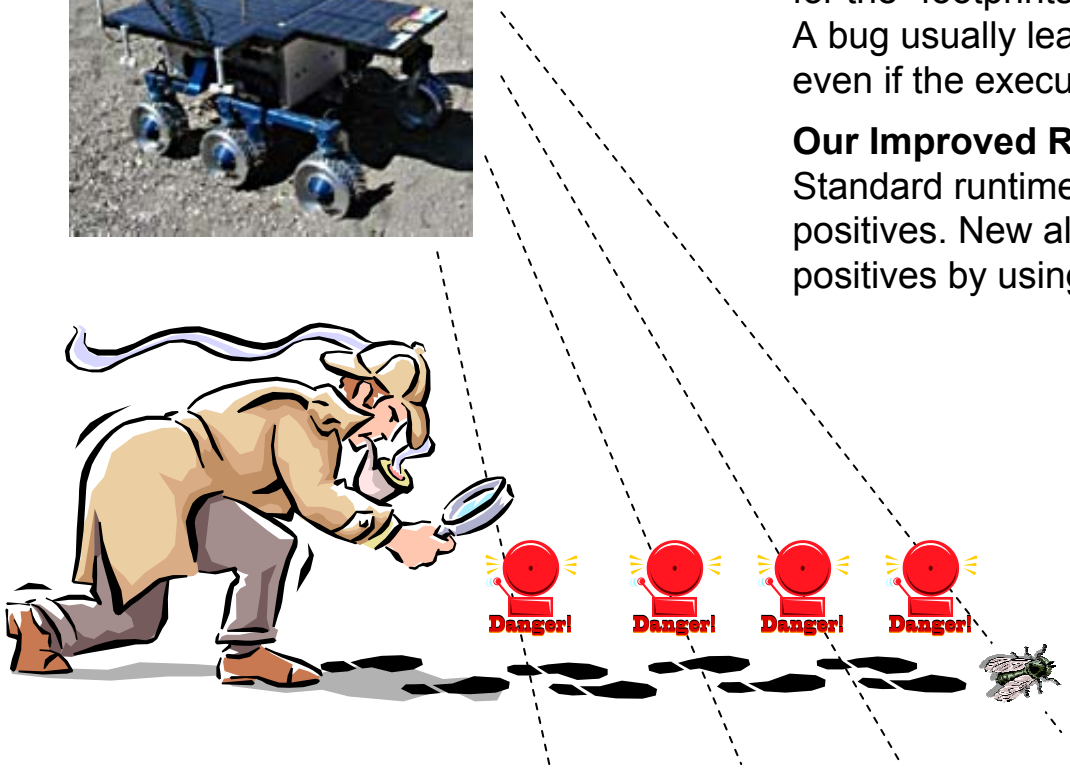
*Issue warnings for all **proper** cycles*

Deadlock Analysis with Fewer False Positives

The Problem of Non-Determinism
Multithreaded software is non-deterministic. Some executions may exhibit a bug, eg. a deadlock, while others may not. Standard testing may therefore not reveal the bug.

The Solution of Runtime Analysis
Runtime analysis examines a single execution trace for the "footprints" of bugs; eg. cycles in a lock graph. A bug usually leaves prints in most execution traces, even if the executions do not exhibit the bug.

Our Improved Runtime Analysis Algorithm
Standard runtime analysis of deadlocks yields false positives. New algorithm reduces number of false positives by using labeled lock graphs.



Case Study results
K9 rover: Found one unexpected deadlock, confirmed one data race, and found all seeded deadlocks and data races.
DS1 Attitude Control System: Found two unexpected data races, and all seeded data races.

Explanation of Accomplishment

- **POC:** Klaus Havelund (ASE group, Code IC, havelund@email.arc.nasa.gov)
- **Background:** Concurrency-related errors in multi-threaded mission software often manifest themselves only by intermittent bugs and hence are difficult to find by testing. Runtime Analysis is a solution; it is a technique that analyzes the trace of a single execution of a program, inferring possible problems in other executions. It scales well to large programs. The standard runtime analysis algorithm detects possible deadlocks. However, it suffers in that it reports many false positives. This requires the user to investigate deadlocks that cannot actually appear, making the technique less usable.
- **Accomplishment:** We have developed an enhanced runtime analysis algorithm for deadlock detection that issues fewer false positives. It reduces the problem of finding deadlocks to finding a cycle in a labeled graph that describes the lock hierarchies appearing during execution. This algorithm, and a data race detection algorithm, have been implemented in the Java PathExplorer (JPaX) runtime verification tool. JPaX has been applied to two major case studies. The K9 rover developed at Ames was analyzed after having been seeded with deadlocks. All deadlocks were found. An early version of the algorithm found an unexpected deadlock in the K9 rover. JPaX has also been applied to the Deep-Space 1 attitude control system. This system was cycle free, but two unexpected data races were identified.
- **Future Plans:** We are currently extending the capability of JPaX to be able to detect other kinds of concurrency errors. Currently we are implementing an algorithm for detecting higher level data races. Errors in the Remote Agent found by the POC were caused by such higher level data races. We are also extending the tool to find other forms of deadlocks, also referred to as communication deadlocks. JPaX furthermore includes a capability for checking conformance of an execution trace with a user provided requirements document. Future work includes improving this framework.